MICROCOPY RESOLUTION TEST CHART
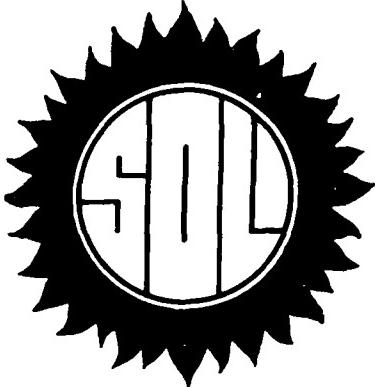
NATIONAL BUREAU OF STANDARDS 1963 A

AD-A170 396

APJ 2/592.

Systems
Optimization
Laboratory

MAINTAINING LU FACTORS OF A GENERAL SPARSE MATRIX

by

Philip E. Gill, Walter Murray
Michael A. Saunders and Margaret H. Wright

OTIC FILE COPY

DTIC
ELECTE
JUL 2 3 1986
E

Department of Operations Research
Stanford University
Stanford, CA 94305

86 7 23 379

SYSTEMS OPTIMIZATION LABORATORY
DEPARTMENT OF OPERATIONS RESEARCH
STANFORD UNIVERSITY
STANFORD, CALIFORNIA 94305-4022

MAINTAINING LU FACTORS OF A GENERAL SPARSE MATRIX

by

Philip E. Gill, Walter Murray
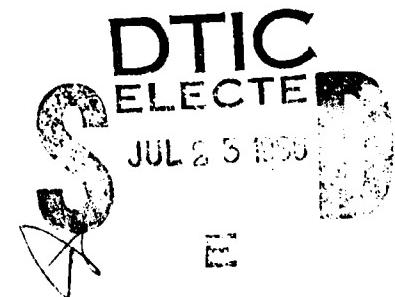Michael A. Saunders and Margaret H. Wright

TECHNICAL REPORT SOL 86-8

May 1986

Maintaining *LU* Factors of a General Sparse Matrix *

Philip E. Gill, Walter Murray,
Michael A. Saunders and Margaret H. Wright

Systems Optimization Laboratory
Department of Operations Research
Stanford University
Stanford, California 94305

Technical Report SOL 86-8
May 1986

Abstract

We describe a set of procedures for computing and updating an *LU* factorization of a sparse matrix A, where A may be square (possibly singular) or rectangular. The procedures include a Markowitz factorization and a Bartels-Golub update, similar to those of Reid (1976, 1982). The updates provided are addition, deletion or replacement of a row or column of A, and rank-one modification. (Previously, column replacement has been the only update available.)

Various design features of the implementation (LUSOL) are described, and computational comparisons are made with the LA05 and MA28 packages of Reid (1976) and Duff (1977).

Keywords: Sparse matrix, *LU* factors, matrix factorization, matrix updates, rank-one modification, Fortran software.

---

* We dedicate this paper to Dr. James H. Wilkinson in recognition of his profound influence on *LU* factorization, and in the belief that $\begin{pmatrix} 1 & \\ -\mu & 1 \end{pmatrix}$ and $\begin{pmatrix} 1 & -\mu \\ & 1 \end{pmatrix}$ are his very favorite matrices.

1. Introduction

Gaussian elimination has long been used to obtain triangular factors of a matrix A. We write the factorization as $A = LU$, where L and U are nominally lower and upper triangular. In general, the rows and columns of L and U need to be reordered to make them strictly triangular. If A is rectangular, U is upper trapezoidal.

The usual application of LU factors is to the solution of linear equations $Ax = b$. Often just one such system is be solved, but in many applications there is a sequence of related systems, in which A is subject to certain elementary changes. We describe a set of procedures designed for both cases. The procedures are grouped according to the following major functions:

Factor For a given $m \times n$ real, sparse matrix A, use some form of Gaussian elimination to compute a factorization $A = LU$, where L is $m \times m$ and U is $m \times n$.

Solve For a given m-vector b, use the LU factors to find an n-vector x that solves the linear system $Ax = b$. (If A is singular or rectangular, only a subset of the equations may be satisfied accurately.)

Update Modify L and U to obtain a new factorization $A = LU$ when A is altered in one of the following ways:

> addition, deletion or replacement of a column of A;
>
> addition, deletion or replacement of a row of A;
>
> modification by a matrix of rank one ($A \leftarrow A + \sigma v w^T$).

Each *Update* maintains U as an explicit, sparse, permuted triangle, but L is held in *product form*, as the product of an arbitrary number of triangular matrices. The properties of the LU factors are as follows:

1. $L = M_1 M_2 M_3 \ldots$ is a product of unit triangular matrices M_k, where each M_k is the identity matrix with just one nonzero entry $-\mu_k$ above or below the diagonal. Thus,

$$M_k = I - \mu_k e_{i_k} e_{j_k}^T \tag{1.1}$$

for some unit vectors e_{i_k}, e_{j_k}, $i_k \neq j_k$. The scalars μ_k are called *multipliers*.

2. The multipliers in (1.1) are *bounded* according to

$$|\mu_k| \leq \bar{\mu}, \tag{1.2}$$

for a given *threshold* $\bar{\mu} \geq 1$. A typical value is $\bar{\mu} = 10$, which allows a balance between numerical stability and the preservation of sparsity (see Section 2).

3. The matrix PUQ is upper triangular for certain permutation matrices P and Q.

Note that an LU factorization of this kind exists for any matrix A, whether square or rectangular, singular or not. By construction, L is nonsingular and should be well-conditioned throughout as long as $\bar{\mu}$ in (1.2) is not too much greater than 1. The dimensions and condition of A are almost always reflected in U, which will be singular if A is singular.

The procedures to be described have been implemented in a set of Fortran routines called LUSOL. We shall use that name to refer to some or all of the complete set.

1.1. Background. A method for updating LU factors following column replacement was proposed by Bartels and Golub (1969), who later suggested a more efficient approach in which matrices M_k were accumulated in product form as above (see Bartels, 1971; Bartels, Stoer and Zenger, 1971). The freedom to choose between the two forms of M_k ($i_k < j_k$ or $i_k > j_k$) was fundamental. The multipliers were bounded by $\bar{\mu} = 1$, since sparsity was not a consideration.

The techniques to be discussed are most closely related to those developed by Reid (1976, 1982) in the subroutine package LA05. LA05 works with square matrices that are subject to column replacement; it performs a sparse LU factorization and a sparse Bartels-Golub update. Properties 1–3 above are maintained during the update, and in practice the package has proved to be efficient and reliable. A potential drawback is that Property 2 is not necessarily satisfied by the initial factorization (which is stabilized by controlling the size of the elements of U, rather than those of L). If A is significantly ill-conditioned initially, some of the multipliers μ_k will probably be large. Since these are retained for all subsequent updates, the condition of the factors of L cannot improve even if A later becomes well-conditioned.

In developing LUSOL, our aims have been

- to allow for singular and/or rectangular systems;
- to expand the range of update options, including ones that alter the size of A;
- to ensure stability by controlling the size of the multipliers throughout.

1.2. Applications. The most important update is column replacement, which is vital to the simplex method for linear programming (Dantzig, 1963) and to the reduced-gradient method for linearly constrained optimization (Wolfe, 1962). LUSOL is employed for these purposes within the large-scale optimization code MINOS 5.0 (Murtagh and Saunders, 1983). Column replacement is required in several other algorithms in mathematical programming—notably, methods for solving complementarity problems, and fixed-point algorithms for solving nonlinear equations.

Since LUSOL is unique in allowing A to be rectangular, an application that is likely to increase in importance is the solution of sparse linear least-squares problems

$$\min \|b - Ax\|_2^2, \tag{1.3}$$

in cases where it is practical to compute LU factors of A but not orthogonal factors. Problems of the form (1.3) occur at every iteration in recently proposed "nonlinear" approaches to linear programming (see, e.g., Karmarkar, 1984; Gill et al., 1985; Todd and Burrell, 1985). When $m > n$, the factors of A take the form $A = LU = \hat{L}\hat{U}$, where $\hat{L}$ is $m \times n$ and $\hat{U}$ is $n \times n$. If $\hat{U}$ is nonsingular, the solution of (1.3) can be obtained from the system

$$\min \|b - \hat{L}y\|_2^2, \tag{1.4}$$

where $\hat{U}x = y$. As noted by Peters and Wilkinson (1970), it may be advantageous to solve (1.4) rather than (1.3) if $\hat{L}$ is better conditioned than A—one of the aims of our procedures. For

example, if (1.4) is solved by an iterative algorithm such as the method of conjugate gradients, the rate of convergence of the iterative algorithm may be improved (see Björck, 1976). Some experiments along these lines have been described by Saunders (1979). The linear programming context is discussed by Gill *et al.* (1986).

A further application is to the estimation of the singular values of a sparse matrix. The approach described by Foster (1986) requires a well-conditioned factor L and various column updates.

A review of alternative updating methods for sparse matrices has been given by Gill *et al.* (1984).

2. Fundamentals

The key to Gaussian elimination and to the algorithms described here is the LU factorization of a matrix consisting of two rows. The LU factors take two possible forms:

$$\begin{pmatrix} \alpha & v^T \\ \beta & w^T \end{pmatrix} = \begin{pmatrix} 1 & \\ -\mu_1 & 1 \end{pmatrix} \begin{pmatrix} \alpha & v^T \\ 0 & \bar{w}^T \end{pmatrix}, \qquad \mu_1 = -\beta/\alpha, \qquad \bar{w} = w + \mu_1 v, \qquad (2.1)$$

$$\begin{pmatrix} \alpha & v^T \\ \beta & w^T \end{pmatrix} = \begin{pmatrix} 1 & -\mu_2 \\ & 1 \end{pmatrix} \begin{pmatrix} 0 & \bar{v}^T \\ \beta & w^T \end{pmatrix}, \qquad \mu_2 = -\alpha/\beta, \qquad \bar{v} = v + \mu_2 w. \qquad (2.2)$$

Only the case $\alpha \neq 0$, $\beta \neq 0$ need be considered. (If α or β is already zero, the original matrix is trapezoidal and we regard it as already triangularized; no factorization is needed.)

The choice between the two forms must be made first on numerical grounds, using the threshold $\bar{\mu}$ in (1.2). If $|\mu_1| \leq \bar{\mu}$, (2.1) is regarded as acceptable. Otherwise it must be true that $|\mu_2| \leq \bar{\mu}$, and so (2.2) is acceptable. In effect, (2.2) involves a *row interchange*.

Use of these elementary factorizations is commonly known as *pairwise pivoting*. When $\bar{\mu} > 1$, an associated term is *threshold pivoting*.

2.1. Stability versus sparsity.
If $|\mu_1|$ and $|\mu_2|$ are both less than $\bar{\mu}$, it might seem desirable to choose the smaller of the two, to cater even further to numerical stability. (Recall from standard linear algebra that a power of $\bar{\mu}$ occurs in the bound on the growth in elements of the LU factorization.) However, we follow common practice in making the choice on sparsity grounds, since substantial growth almost never occurs in practice. In general the vectors $(\alpha \quad v^T)$ and $(\beta \quad w^T)$ will be two rows of a sparse matrix, and the data structure used will include a count of the number of nonzeros in each row. Let $len(v)$ (the "length" of v) denote the number of nonzeros in the vector v. If $len(v) \leq len(w)$ we choose (2.1), because the number of nonzeros in the trapezoidal factor will then be minimized. (Only v and w need be considered since $len(\bar{v}) = len(\bar{w})$, even if there is cancellation in forming $\bar{v}$ and $\bar{w}$.)

2.2. Error analysis.
Let ϵ be the precision of floating-point arithmetic and consider the factors in (2.1). For reasonable values of $\bar{\mu}$, the computed μ_1 and $\bar{w}$ are exact for perturbed data $\beta + \delta_\beta$

and $w_j + \delta_j$, where

$$|\delta_0| \le \epsilon|\beta|,$$

$$|\delta_j| \le 2.01\epsilon|\bar{w}_j| \le 2.01\epsilon(|w_j| + |\mu_1||v_j|)$$

(see Wilkinson, 1965; Reid, 1971). Although the relative perturbations δ_j/w_j may not be small, enforcing a bound $|\mu_1| \le \bar{\mu}$ helps avoid excessive *absolute* error, and also discourages a compounding effect when $\bar{w}$ plays the role of v or w in later elementary factorizations.

Pairwise pivoting may be used in many ways to obtain a factorization $A = LU$ for a general matrix A. In particular, Gaussian elimination with partial pivoting may be regarded as pairwise pivoting with the restriction that L be a (permuted) triangle. The classical error analysis of Wilkinson (1965, p. 214) applies when $\bar{\mu} = 1$. This has been generalized by Reid (1971) for arbitrary $\bar{\mu}$. Reid's analysis applies to our Markowitz factorization procedure LU1FAC (see Section 5).

Two other forms of pairwise pivoting to obtain $A = LU$ are described by Wilkinson (1965, pp. 236–239). They are respectively column-oriented and row-oriented but are algebraically equivalent. An error analysis has been given by Sorensen (1984) for the case $\bar{\mu} = 1$. Our second factorization procedure LU2FAC is a threshold form of the row-oriented algorithm (see Section 5.6), and Sorensen's analysis could be extended to cover this case.

Further pairwise pivoting is employed by all of our update procedures. The case of column replacement has been analyzed by Bartels (1971), again assuming $\bar{\mu} = 1$, and an arbitrary sequence of updates might be analyzed in a similar way.

Note, however, that in all of the analyses cited, the error bounds obtained are extremely pessimistic. Suffice to say that threshold pairwise pivoting limits the *likelihood* of growth in $\|L\|$ and $\|U\|$, and with reasonable values such as $\bar{\mu} = 10$ or perhaps 100, it is in practice an effective strategy for factorizing and updating alike.

2.3. Discussion. The factorization (2.1) could have been written in the slightly simpler form

$$\begin{pmatrix} \alpha & v^T \\ \beta & w^T \end{pmatrix} = \begin{pmatrix} 1 & \\ \mu_1 & 1 \end{pmatrix} \begin{pmatrix} \alpha & v^T \\ 0 & \bar{w}^T \end{pmatrix}, \qquad \mu_1 = \beta/\alpha, \qquad \bar{w} = w - \mu_1 v,$$

but we (arbitrarily) prefer addition to subtraction in forming $\bar{w}$, and during forward or backward substitution with the 2×2 triangular factor. Similar remarks apply to (2.2).

Alternatively, it is common to think of Gaussian elimination as *multiplying* by unit triangular matrices rather than factorizing. Thus (2.1) is equivalent to

$$\begin{pmatrix} 1 & \\ \mu_1 & 1 \end{pmatrix} \begin{pmatrix} \alpha & v^T \\ \beta & w^T \end{pmatrix} = \begin{pmatrix} \alpha & v^T \\ 0 & \bar{w}^T \end{pmatrix}, \qquad \mu_1 = -\beta/\alpha, \qquad \bar{w} = w + \mu_1 v,$$

and the factorization $A = LU = M_1 M_2 M_3 \ldots U$ is equivalent to $NA = \ldots N_3 N_2 N_1 A = U$, where N_k is identical to M_k except for the sign of μ_k. We prefer to work with $A = LU$ rather than $NA = U$ because, following a direct *Factor* operation, the quantities (μ_k, i_k, j_k) defining each M_k have the sparsity pattern of an explicit triangular matrix L. The product $N = \ldots N_3 N_2 N_1$, if formed explicitly, would in general be considerably less sparse than L.

3. Data structures

In later sections we shall make explicit reference to the data structures used to represent L and U. Here we define the main data structures—the simpler ones first.

3.1. The permutations P and Q. Recall that the matrix PUQ is upper triangular (or upper trapezoidal). The permutations are represented by two integer arrays P and Q of length m and n, respectively. The k-th diagonal element of PUQ is contained in row P_k and column Q_k of the matrix U.

3.2. Data structures for *Solve* and *Update*. The *Solve* and *Update* routines work with data structures similar to those used by Reid in **LA05**:

1. The components of L are stored in a *sequential file* as a lengthening list of triples (μ_k, i_k, j_k) (one triple for each triangular factor M_k).

2. The nonzeros of U are stored by rows in a *row list* that allows for fill-in (additional nonzeros) when a multiple of one row is added to another. (Reid additionally maintains a column list for the sparsity pattern of U; see Section 3.3.)

The L-file is implemented using three parallel arrays, with entries made backwards, starting at the end. We shall say that L is stored in an *ordered list* $\{A, indc, indr\}$, where A is an array containing the sequence of multipliers μ_k, and *indc*, *indr* are arrays of the corresponding indices i_k, j_k.

Similarly, the U-file is implemented as a *row list* $\{A, indr, lenr, locr\}$ holding pairs (U_{ij}, j), where for $i = 1$ to m the i-th row of U contains $lenr(i)$ nonzeros, stored consecutively in the arrays A and *indr*, starting at location $locr(i)$. The nonzeros in row i are not in any particular order, except the *first* nonzero is normally the i-th diagonal element of PUQ. (Note that *indr* refers to *indices in a row list*, which are *column* numbers, not row numbers.)

When a row of U is modified, we attempt to do the modification in-place, making use of any free space that may have arisen at the end of the row. (We do not look for possible free space before the beginning of the row.) If there is too much fill-in, the row is moved to the end of the row list, where there will generally be ample storage, and the locations previously occupied are marked as free. Thus, the rows of U are not in any particular order, but we know where each row begins and how long it is.

Periodically the row list is *compressed* to recover the free space that accumulates between rows. Compressions do not alter the ordering of rows or nonzeros, but they require a traverse of the entire list; hence the desire to update in-place. Occasionally we force a compression, not because storage is exhausted but because the length of the list is currently l times greater than it would be if compressed (where l is typically 3). This policy should be beneficial in a virtual-storage environment.

3.3. Data structures for *Factor*. The factorization procedure of Section 5.1 requires more complex data structures, to allow efficient searching of both rows and columns of the submatrix

remaining to be factorized. A row list $\{indr, lenr, locr\}$ is used as before to store the column indices of the nonzeros in each row (but not the corresponding elements of U). The nonzeros themselves are stored as pairs (U_{ij}, i) in a column list $\{A, indc, lenc, locc\}$ in order to facilitate the stability test, which compares a potential pivot element with other nonzeros in the same column. Additional data structures maintain the rows and columns in order of increasing length.

Eventually, the columns of L and the rows of U are repacked into the data structures required by the *Solve* and *Update* procedures.

4. Sequences of eliminations

The LU factorizations in (2.1) and (2.2) are constructed to eliminate a single nonzero element α or β from a two-row matrix. In general, Gaussian elimination is organized so that *sequences of consecutive nonzeros* are eliminated from either *one row* or *one column* of a larger matrix. We distinguish between the two cases.

4.1. Forward sweeps. During updates, the matrix PUQ is often upper triangular except for one row (commonly called a *spike*). The process of obtaining LU factors of such a matrix is called a *forward sweep*. Consider the 8×8 example

$$
PUQ = \begin{pmatrix}
x & & & & & & x & \\
 & \alpha & & v_4 & & v_6 & & \\
 & & x & & & & & \\
 & & & x & & x & x & \\
 & & & & x & & & \\
 & & & & & x & & \\
 & \beta & & w_4 & & & w_8 & \\
 & & & & & & & x
\end{pmatrix},
$$

where the spike is row 7. Since the first nonzero in the spike row (β) lies in column 2, the first stage of the forward sweep is to factorize rows 2 and 7 to eliminate either β or α, using (2.1) or (2.2).

Note that many of the rows of U will have been used previously to eliminate various nonzeros. Hence it is likely that the existing diagonals of PUQ will be sufficiently large to eliminate nonzeros in the spike row without interchanges. Our strategy therefore is to *treat the spike row specially* by constructing a vector of pointers to each of its nonzeros (in this case β, w_4 and w_8). For convenience, let $w_2 \equiv \beta$; then we define an n-vector *locw* as follows:

$$
locw(j) = \begin{cases} l & \text{if } w_j \text{ is stored in location } l \ \ (w_j \neq 0, \, l > 0); \\ 0 & \text{otherwise (i.e., if } w_j = 0). \end{cases}
$$

To perform the elimination, *locw* is used to scan the spike row (whose index is denoted by iw), looking for the next nonzero. Assuming that no row interchanges are required, the outer

loop has the following form, where k_1 and k_2 mark the beginning and diagonal of the spike row respectively, and *last* marks the end of the list of nonzeros in w:

for $k = k_1$ to k_2
    $j = Q_k$, $l = locw(j)$
    if $l > 0$ then
        $iv = P_k$ (α is in row iv)
        $\alpha = A(locr(iv))$ (first nonzero in row iv)
        $\beta = A(l)$
        $\mu = -\beta/\alpha$
        (delete β from w):
            $A(l) = A(last)$, $jlast = indr(last)$
            $indr(l) = jlast$, $indr(last) = 0$
            $locw(jlast) = l$, $locw(j) = 0$
            $last = last - 1$
        (inner loop):
            compute $w \leftarrow w + \mu v$
    end if
end outer loop

If a row interchange is needed to satisfy the stability test, P_k is set to iw and P_{k_2} to iv. We exit the loop and alter *locw* to mark the nonzeros of the new w (i.e., the old v). We then re-enter the loop from the top with k_1 set to the existing value of k, knowing that the opposite interchange will not occur.

In the inner loop, the nonzeros of v are scanned (using the row list). The array *locw* now determines whether a new nonzero will be created in w. If w_j is already nonzero, its location is known, and it can be modified in-place. Otherwise, a fill-in occurs, and we insert the new element into the row list at the end of the present elements of w. Assuming space for fill-in, the inner loop has the following form:

for j such that $v_j \neq 0$
    $l = locw(j)$
    if $l > 0$ then
        $A(l) = A(l) + \mu v_j$ (modify existing w_j)
    else
        $last = last + 1$ (add fill-in to the end of w)
        $A(last) = \mu v_j$
        $indr(last) = j$
        $locw(j) = last$
    end if
end inner loop

If w is currently stored at the end of the row list, there will be space for any amount of fill-in. Otherwise, an identical inner loop is used, except following the "else" we test whether the location about to be used for the fill-in is already occupied (by the first nonzero of some other row). If so, we exit the loop, move w to the end of the row list, and continue with the simpler inner loop.

Since the outer loop deletes the current β from w at each : age, there is always room for at least one fill-in during the inner loop. In the above example, suppose that the stability test does not force a row interchange. When β is eliminated, it is overwritten by the last element of w (which could be β, w_4 or w_8 since the nonzeros in each row are not in any particular order). The location previously occupied by the last element is then free to accommodate the fill-in caused by v_6. Note that $locw(6) = 0$ initially but $locw(4)$ points to w_4, which can be modified in-place. After the fill-in, $locw(6)$ will point to the new nonzero w_6. On conclusion of the first part of the forward sweep, a new triple $(\mu, 7, 2)$ is added to L.

Continuing with the present example, the second part of the forward sweep is the same as the first, with the modified w_4 playing the role of β and row 4 becoming the current $(\alpha \quad v^T)$. This time the fill-in produces a nonzero w_7, which by chance will survive the remainder of the sweep to become a diagonal element of the final PUQ. A triple $(\mu, 7, 4)$ is also added to L. Finally, the third part of the sweep adds a triple $(\mu, 7, 6)$ to L, eliminating w_6 but not altering w_7 or w_8.

When the outer loop terminates, it remains to set previously non-zero elements of $locw$ to zero (in preparation for any future sweep), and to move the diagonal of the spike row to the front of that row in the row list. If the diagonal was never created or has vanished as a result of cancellation, we return an indication of singularity. This can often be ignored since subsequent updates may remove the singularity before any solve is requested.

In practice we find that forward sweeps usually require very little rearrangement of the rows of U. When $\bar{\mu} = 10$, the stability test seldom forces a row interchange, and the sparsity test applies with not much greater frequency. Once a spike row *has* been moved to the end of the row list, it is likely to remain there for the rest of the forward sweep. The increase of nonzeros in L and U is relatively slight because almost all of the rows of U are completely unaltered.

4.2. Backward sweeps. As just described, a forward sweep involves adding a multiple of several different rows of U to one particular (spike) row. In contrast, a backward sweep involves adding a multiple of a spike row to several other rows of U, so that the potential for fill-in is much greater.

In particular, suppose that $A = LU$ with PUQ triangular as usual, and that we wish to obtain an LU factorization of the matrix $(c \quad U)$, where c is a given sparse vector. To illustrate

the procedure, consider the example

$$
(\, Pc \quad PUQ \,) = \begin{pmatrix}
c_1 & x & & & & & x \\
\alpha & & v_2 & v_4 & v_6 & & \\
& & x & & & & \\
& & x & x & x & & \\
& & & x & & & \\
& & & x & & & \\
\beta & & & & & w_7 & w_8 \\
& & & & & & x
\end{pmatrix},
$$

where it is convenient to assume that $P = I$ and to let $\alpha \equiv c_2$, $\beta \equiv c_7$. The approach is to process the nonzeros of Pc backwards, using the bottom nonzero β to eliminate "higher" nonzeros one by one. In this situation, it is desirable to use (2.2) whenever possible, since adding a multiple of a "short" row w to earlier rows of PUQ will not create subdiagonal elements that would otherwise have to be eliminated. With the present example, we hope to factorize rows 2 and 7 and then rows 1 and 7 to eliminate α and c_1 respectively.

As always, it may be necessary to use (2.1) instead. In the example, adding a multiple of v would turn w into a row spike of PUQ. However, we effectively perform a row interchange by switching two elements of P, so that the modified w becomes a normal row of U and v becomes the spike.

In general, the bottom nonzero β defines the current spike row, and each $2 \times (n+1)$ factorization (2.2) adds a multiple of the spike w to another row that is at least as long, so that no new subdiagonal elements are introduced into PUQ. If (2.1) is used, the spike is in effect redefined to be a longer row, but again will be added only to rows that are at least as long.

When the backward sweep is complete, we have a factorization $(\, c \quad U \,) = \tilde{L}\tilde{U}$ where $\tilde{U}$ is a permuted trapezoid. More importantly, however, the last n columns of $\tilde{U}$ are triangular except for at most one row. To complete an *Update*, some other vector is typically added to the spike row, and the resulting matrix is triangularized by a single forward sweep.

To implement the backward sweep we again treat the spike row specially (as in a forward sweep), since it is likely to remain the spike for several eliminations. The array *locw* points to nonzeros within the spike row as before, and a first inner loop takes the form

> for j such that $v_j \neq 0$
>> $l = locw(j)$
>> if $l > 0$ then
>>> $v_j = v_j + \mu A(l)$ (modify existing v_j using w_j in location l)
>>> $mark(l) = k$
>>> $numw = numw + 1$
>> end if
> end first inner loop

where *mark(l)* is needed to record which nonzeros w_j are accounted for on the k-th pass through the outer loop (the count of these elements is given by *numw*). Each other nonzero of w creates a new element of v, giving a second inner loop of the form:

> for l locating $w_j \neq 0$
> > if *mark(l)* $\neq k$ then
> > > $last = last + 1$ (add fill-in v_j to the end of v)
> > > $A(last) = \mu A(l)$
> > > $indr(last) = j$
> > end if
> end second inner loop

The maximum amount of fill-in, $len(w) - numw$, indicates whether the second inner loop is to be performed, and if so, whether the number of free locations at the end of v is adequate. When necessary, the row containing v is moved to the end of the row list before the second loop is executed.

The data structures described for a backward sweep are similar to those used in the Markowitz *Factor* routine MA28 (Duff, 1977) and in our own Markowitz *Factor*, LU1FAC (see Section 5.1).

In the implementation, the inner loops for forward and backward sweeps are slightly more complex than shown. Following common practice, we test computed "nonzeros" against some absolute tolerance (typically of order 10^{-12} on machines with 15 or more digits of precision) and if possible eliminate them immediately from the data structure. Although numerical cancellation is rather rare, experience shows that taking advantage of it can improve the sparsity of the factors slightly and at the same time reduce the occurrence of floating-point underflow.

4.3. Sparse AXPY procedures. The vector operation known as AXPY ($y \leftarrow ax + y$) usually involves a single scalar a and two dense vectors x and y. It can be generalized in various ways. For example, there could be a *sequence* of AXPYs involving several x's and one y (as in $y \leftarrow y + Xa$ where a is now a vector), or several y's and one x (as in $Y \leftarrow Y + xa^T$); see the procedures GEMV and GER1 of Dongarra *et al.* (1985). Also, some of the vectors could be *sparse*; see AXPYI in Dodson and Lewis (1985).

We note that the forward sweep described in Section 4.1 is in some sense a sparse implementation of GEMV, since it takes the form $w \leftarrow w + U^T\mu$ if no interchanges occur, or a sequence of such operations otherwise, where μ is a vector of multipliers. Taking the opposite view, a general sparse GEMV would perhaps be useful for implementing a forward sweep. However, the vector μ is not known in advance, and it is important to be able to interrupt the process for stability and sparsity reasons and to continue efficiently with some other w. (In fact it is more accurate to view a forward sweep as solving the system $U^T\mu = w$ by forward substitution, with provision for swapping the partially transformed right-hand side w with some other column of U^T, in order to restrict the size of the solution vector μ.)

Similarly, the backward sweep is a sparse form of GER1, since it takes the form $U \leftarrow U + \mu w^T$ (or a sequence of such operations). In this case μ and w (or a sequence of such quantities) can be determined in advance.

5. Factorization procedures

5.1. A Markowitz factorization.
We now describe a procedure (LU1FAC) that computes a factorization $A = LU$ by Gaussian elimination with row and column interchanges, using the pivotal strategy due to Markowitz (1957) to choose permutations P and Q such that PUQ is upper triangular. Many of the implementation techniques follow those used by other authors (notably Reid in LA05, Duff in MA28, and Zlatev *et al.* in Y12M), but a few novel features are noted below.

First recall that the Markowitz merit function for selecting A_{ij} as the next diagonal of PUQ is $M_{ij} = \big(lenr(i) - 1\big)\big(lenc(j) - 1\big)$, and the strategy is to choose as small an M_{ij} as possible subject to a stability test. For $k = 1$ to $\min(m, n)$, the k-th stage of the factorization computes the k-th column of L and the k-th row of PUQ, and updates A_{ij}, $lenr(i)$ and $lenc(j)$ appropriately (leaving a submatrix with one less row and column).

Various strategies have been proposed to limit the search for pivots, and to break ties when M_{ij} is the same for several potential pivots A_{ij}. The usual first step to limit searching is to keep track of the rows and columns with fewest nonzeros (Curtis and Reid, 1971). In LU1FAC the rows and columns are held in two separate ordered lists within the permutation arrays, with the shortest ones appearing first. Thus, the rows of length nz are the set $\{P_l\}$ for $l = iploc(nz)$ to $iploc(nz + 1) - 1$, where $iploc(i)$ gives the location in P of the first row with i nonzeros (and similarly for the columns). Other authors have used linked lists rather than ordered lists, but to date we have preferred the slight saving on integer workspace. By arranging to have available both the old length and the new length of a row or column, we can update a list quite efficiently by "bubbling" the row or column up or down from one set to the next. (The length of a row or column seldom changes by more than one or two at each stage of the elimination, and frequently does not change at all.)

Following Reid (1976), we search columns of length 1, then rows of length 1, then columns of length 2, and so on, applying a stability test to each A_{ij} encountered. The search is terminated when all remaining rows and columns are clearly too dense to yield an improved pivot.

5.2. Curtailing the search.
To place a definite bound on the effort involved, some authors search only the p shortest rows and no columns (or the p shortest columns and no rows), where p is an input parameter; see Østerby and Zlatev (1983). Setting p as low as 1, 2 or 3 (say) can lead to considerable savings on certain regularly structured matrices.

In the present context, one could search only the p shortest rows and the q shortest columns. However, the real aim is to economize when there are many "ties"—i.e., when perhaps hundreds of nonzeros all have the same merit and satisfy the stability test. We therefore terminate the

search when p consecutive ties have been encountered. Although some arbitrariness remains in the choice of p, this strategy has the advantage of continuing the search as long as improved merits are being found with a reasonable (specified) frequency.

5.3. Breaking ties. Let A_{ij}, M_{ij} and μ denote a nonzero, its merit, and the largest multiplier generated if it were eventually chosen to be the pivot, and let A_{best}, M_{best} and μ_{best} denote those quantities for the best pivot found so far.

A new candidate will be rejected if $M_{ij} > M_{best}$ or $\mu > \bar{\mu}$, and will be accepted as the new "best" pivot if $M_{ij} < M_{best}$ and $\mu \le \bar{\mu}$. Otherwise, $M_{ij} = M_{best}$ and $\mu \le \bar{\mu}$, and some tie-breaking strategy is required. We now discuss the following possibilities:

TB1: Ignore ties, as in LA05.

TB2: Maximize $|A_{ij}|$, as in Y12M.

TB3: Minimize the maximum multiplier μ, as in MA28 (revised 1983).

TB4: A combination of the last two (to be described below).

TB1 has the advantage of allowing the search to terminate earlier (if it is not curtailed by a small p, as discussed in Section 5.2). TB2 has a beneficial effect on diagonally dominant systems with symmetric structure: the pivots chosen will always be diagonal elements, so that symmetry is preserved and stability is assured regardless of the size of $\bar{\mu}$; see Zlatev (1981) and Østerby and Zlatev (1983). The same favorable property can be proved for TB3 and for our particular choice of TB4.

TB3 deals directly with the size of the multipliers, which is particularly important in the context of updating. Like the stability test itself, TB3 requires A_{ij} to be sufficiently large relative to other nonzeros in column j, *excluding itself*. For example, the following algorithm applies the stability test and simultaneously breaks ties, without necessarily scanning all nonzeros in column j:

$$\text{if } M_{ij} < M_{best}, \; tol = |A_{ij}|\bar{\mu}$$
$$\text{if } M_{ij} = M_{best}, \; tol = |A_{ij}|\mu_{best}$$
$$C_{max} = 0$$

```
for r such that A_rj ≠ 0 do
    if r ≠ i then
        C_max = max{C_max, |A_rj|}
        if C_max ≥ tol, exit to end of outer loop (rejecting A_ij)
    end if
end inner loop
```

$$M_{best} = M_{ij}, \qquad A_{best} = |A_{ij}|, \qquad \mu_{best} = C_{max}/A_{best}.$$

This strategy is suitable if there is no concern about singularity or curtailing the search as discussed in Section 5.2.

In order to count consecutive ties, the code above must be modified so that *tol* is always set to $|A_{ij}|\bar{\mu}$. In this case, the exit from the inner loop will not occur quite so often, but we then have the ability to combine TB2 and TB3. For the experiments reported in Section 9, the tie-breaking rule was as follows, with γ set to 2:

> TB4: Favor a small μ as already described for TB3. However, if μ and μ_{best} are both sufficiently small ($\mu \leq \gamma$ and $\mu_{best} \leq \gamma$), then choose the larger pivot.

For example, if the current best pivot is $A_{best} = 0.1$ with $\mu_{best} = 0.001$, a new candidate $A_{ij} = 1000$ is preferred if its maximum multiplier μ is no larger than 2. Even on nonsingular unstructured matrices, rule TB4 appears to have slightly better numerical properties than TB3 alone.

5.4. Singular systems.

On ill-conditioned, singular or rectangular systems, particularly when $m < n$, it is common to hope that "small" elements of PUQ will not occur on the diagonal (except when necessary). For example, of the possibilities

$$PUQ = \begin{pmatrix} 1 & 1 & \\ & 2 & 2 & 2 \\ & & 10^{-6} & 3 \end{pmatrix} \quad \text{and} \quad PUQ = \begin{pmatrix} 1 & 1 & \\ & 2 & 2 & 2 \\ & & 3 & 10^{-6} \end{pmatrix},$$

the second is preferred. Unfortunately, none of the pivoting strategies discussed above satisfies such a preference. The stability test is a threshold version of partial pivoting, whereas the desired effect can be guaranteed only by some form of complete pivoting (e.g., see Wilkinson, 1979).

Although Rule TB4 tends to achieve the desired effect on singular matrices, in order to be *certain* of avoiding unnecessarily small diagonals, it is necessary to know A_{max}, the largest nonzero in the submatrix remaining to be factored. Before any tie-breaking rule is applied, we would then have to reject a potential pivot A_{ij} if it were smaller than δA_{max} for some conservative value such as $\delta = 10^{-3}$. Maintaining A_{max} may prove to be expensive, but we hope to investigate ways of doing so. One possibility is to update a vector containing the largest nonzero in each column (since only a few columns are altered at each stage), and to maintain a permutation array that lists these values in descending order, so that A_{max} will be readily available as the first element.

5.5. The elimination loop.

Once a pivot has been selected, the actual elimination adds a multiple of the pivot column to all other columns containing nonzeros in the pivot row. Most of the arithmetic occurs in an inner loop that alters existing nonzeros (as opposed to creating new ones). The alteration is performed in-place, using an inner loop that is essentially the same as in MA28 (Duff, 1977). Fill-in is handled later in several stages, in a way that avoids calls to the storage-compression routine from within the inner loop—a seemingly desirable feature but not a crucial matter in practice.

5.6. A preassigned factorization.

While the Markowitz strategy performs admirably in practice, it is usual to expect greater efficiency in cases where "good" permutations P and Q are already known. (For example, if a matrix A has already been factorized by LU1FAC, the resulting permutations may be almost acceptable for some other matrix B that has the same sparsity structure but different numerical values.)

Such cases are treated by a procedure (LU2FAC) that calls the forward sweep routine m times, processing each row of PBQ in turn to eliminate any nonzeros below the diagonal. If A and B are identical, essentially the same factorization will be obtained (with L stored by rows instead of columns). Otherwise, the column permutation Q will be retained but the row permutation P will be perturbed where necessary to preserve stability.

The code NSPIV of Sherman (1978) is also a row-oriented implementation of Gaussian elimination that assumes availability of "good" row and column orderings. In contrast to our procedure, NSPIV retains the given P but alters the column permutation Q to preserve stability. Sherman considered several methods for implementing what is effectively a forward sweep as defined here. We have not attempted a detailed comparison. Instead we note that LU2FAC makes use of code already required by the *Update* procedures, and is therefore essentially "free".

For some applications, suitable permutations P and Q could be obtained from the P^3 or P^4 ordering algorithms of Hellerman and Rarick (1971, 1972) or from the P^5 algorithm of Erisman *et al.* (1985). These so-called Preassigned Pivot Procedures reorder a square, unsymmetric, sparse matrix to be close to lower triangular form. (Thus, to factorize a given A, NSPIV would work with the ordering obtained from A itself, but LU2FAC should be supplied with the ordering obtained from A^T.)

In the case of P^5 (transposed), the ordered matrix would be block upper triangular:

$$PAQ = \begin{pmatrix} B_{11} & B_{12} & \ldots & B_{1b} \\ & B_{22} & \ldots & B_{2b} \\ & & \ddots & \vdots \\ & & & B_{bb} \end{pmatrix},$$

and each block would also be block upper triangular except for the bottom rows:

$$B_{ii} = \begin{pmatrix} D_{11} & \ldots & D_{1d} & \\ & \ddots & \vdots & C \\ & & D_{dd} & \\ R & & & S \end{pmatrix}.$$

This structure is well suited to pairwise elimination by rows, since each square diagonal matrix D_{jj} will be triangularized independently of the others, before the bottom rows $(R \quad S)$ are processed. Row interchanges for stability do not disrupt the structure at any stage. The algorithm implemented by LU2FAC therefore provides a stable, explicit factorization that apparently makes maximum use of the P^5 ordering. Since symbolic orderings are typically faster than Markowitz orderings, a future comparison of MA28 or LU1FAC with the combination of P^5 and LU2FAC would be of definite interest.

6. Solve procedures

This section describes procedures to perform solves and matrix multiplication once an LU factorization has been computed (and possibly updated).

6.1. Solution of systems involving A, L and U. Given the current factorization $A = LU$ and a vector y, the procedure LU6SOL computes a solution x to one of the following systems:

$$Lx = y, \quad L^T x = y, \quad Ux = y, \quad U^T x = y, \quad Ax = y, \quad A^T x = y.$$

(The last two cases make use of the first four.)

The solves involving L and L^T can be performed by a short loop running through the list of triples (μ_k, i_k, j_k), as in LA05. In many cases, some components of x are negligible, thereby allowing half of the associated pass through the loop to be skipped.

Recall that L is of the form $L_0 M$, where L_0 is the result of a direct factorization and M is a product of updates (if any). To increase efficiency slightly during solves with L and L^T, the triples corresponding to L_0 are treated specially if they were produced by the Markowitz procedure (but not if they came from LU2FAC). Instead of treating each triple separately as mentioned above, a somewhat more complicated double loop takes advantage of the fact that j_k is constant for each column of L_0. This has the further advantage that if a component of x is negligible, a set of consecutive triples, corresponding to a column of L_0, can be skipped during solves with L (but not during solves with L^T, since the row structure of L_0 is not known).

Solves involving U and U^T are similar to those involving L_0^T and L_0 respectively. They are somewhat different from LA05 because we retain only the row structure of U (whereas LA05 maintains the sparsity pattern of both the rows and the columns). Comparative timings would depend to a large extent on the sparsity of y. The outer loop is complicated in our case by the fact that U could be singular: either the length of a row could be zero, or the first nonzero stored might not lie on the diagonal.

6.2. Associated procedures. In many contexts it is desirable to have a procedure for computing matrix-vector products of the form

$$x = Ly, \quad x = L^T y, \quad x = Uy, \quad x = U^T y, \quad x = Ay, \quad x = A^T y.$$

We give two examples. First, the j-th column of A can be recovered as the product $a_j = LU e_j$ after A has been overwritten by its LU factors. Second, in solving the sparse least-squares problem (1.3), numerous products Ly and $L^T z$ are required during iterative solution of the associated problem (1.4).

Stability and nearness to singularity may be monitored through the following quantities:

- $\sigma_1 = \max |\mu_k|$ and $\sigma_2 = \sum \mu_k^2 / q$ (where q is the current number of nonzeros in L);
- $\max |U_{ij}|$, the largest element in U;
- $\max |U_{kk}|$ and $\min |U_{kk}|$, the largest and smallest nonzero diagonals in U.

For example, if $\sigma_2 \gg 1$, it may be advisable to refactorize with a smaller bound ($\bar{\mu} < \sigma_1$) on the multipliers.

To pinpoint singularities, an n-vector w is computed as follows. Initially w_j is set to be $\max_i |U_{ij}|$, the largest element in the j-th column of U. Let d_j be the diagonal element associated

with this column. If $d_j \leq t_1$ or $d_j \leq t_2 w_j$ (where t_1 and t_2 are input tolerances) then w_j is negated. The "number" of singularities is then the number of non-positive entries in w. This information is useful if a nonsingular factorization is essential. For example, in the simplex method for linear programming, one could replace the offending columns by judiciously chosen unit vectors, using a series of updates.

In order to solve compatible singular systems, the solve procedure bypasses zero diagonals in U, setting the associated elements of x to zero and summing the residuals on the corresponding equations. (Here we follow Duff, 1977.)

7. Tools for the update procedures

All seven update procedures (to be described in Section 8) use a forward sweep, and the last three use a backward sweep. We now describe three other tools needed in the updates.

7.1. Removing and inserting a column of U. When the j-th column of A is replaced or deleted, any nonzeros in the j-th column of U must be removed. Because we do not maintain the column structure of U, a substantial number of rows of U need to be scanned by our procedure, whereas the analogous part of LA05C (Reid, 1976, 1982) may examine relatively few. However, the inner loop to find the unwanted nonzeros is a single statement

$$\text{if } indr(i) = j \text{ then exit loop.}$$

Furthermore, by scanning the rows in pivotal order (P_k, for $k = 1$ to $\min(m, n)$), we can terminate as soon as $Q_k = j$, thereby setting an index k that is required anyway.

Conversely, when a column is replaced or added, a sparse vector v must be inserted as the j-th column of U. Again the first $\min(m, n)$ rows are examined in pivotal order, looking for nonzeros of v to insert, and an index l is returned to mark the last nonzero found. If the row list has been compressed recently, a substantial number of rows may need to be moved to the end of storage before the elements of v can be inserted.

7.2. Eliminating a single column. The final basic ingredient is a *column elimination* procedure, which is needed when A has more rows than columns. The first $\min(m, n)$ elements of v will have been processed by the column insertion procedure just described, and the remainder are treated as shown by the following example.

Consider triangularizing a matrix of the form

$$PUQ = \begin{pmatrix} x & & & x & x \\ & x & & x & \\ & & x & & x \\ & & & x & \\ & & & & x \\ & & & & v_6 \\ & & & & v_7 \\ & & & & \\ & & & & v_9 \end{pmatrix}$$

in which $m = 10$, $n = 5$ and just one column has nonzeros v_i below the main triangle. While packing these nonzeros into the L-file, we note which is the largest—say v_p, where $|v_p| = \max_{i>n} |v_i|$. This pivot element is overwritten with the last packed nonzero, and the other packed elements are changed to $-v_i/v_p$ to become the appropriate multipliers. The matrix U is then triangular except for a single element in row p, which is eliminated by a forward sweep.

8. Update procedures

Here we describe the seven available *Update* procedures. In each case, the modified matrix A will be denoted by $\bar{A}$.

8.1. Replacing a column (the Bartels-Golub update).

Suppose that the j-th column of A is replaced by a given vector c:

$$\bar{A} = A + (c - a_j)e_j^T. \tag{8.1}$$

Column replacement, the prototype *Update*, is the only update procedure that alters the column permutation Q in order to improve sparsity. (Using the general rank-one update of Section 8.7 for the special case (8.1) does not alter Q and would tend to be less efficient.)

The first step is to solve $Lv = c$ (Section 6.1). The existing j-th column of U is then removed and v is inserted as a new sparse j-th column (Section 7.1), yielding a modified matrix $\tilde{U}$ and two indices k and l. At this stage we have $\bar{A} = L\tilde{U}$, where $P\tilde{U}Q$ is upper triangular except for its k-th column, whose last nonzero is in row l ($1 \le k \le l \le \min(m,n)$), not counting elements below the main triangle.

Now suppose that Q is altered by a cyclic permutation that moves its k-th column into position l and shifts the intervening columns one place to the left, giving a new column ordering $\tilde{Q}$. Most descriptions of the Bartels-Golub update refer to an *upper Hessenberg* matrix, which would be $P\tilde{U}\tilde{Q}$ in this notation. However, it is more useful to apply the same cyclic permutation to the rows of P, giving a new row ordering $\tilde{P}$ such that $\tilde{P}\tilde{U}\tilde{Q}$ is upper triangular except for its l-th row (and perhaps the bottom of its l-th column).

For example, when $m = 10$, $n = 7$, $k = 2$ and $l = 5$, we have

$$\tilde{P}\tilde{U}\tilde{Q} = \begin{pmatrix} x & . & . & l & k & . & . \\ & x & & l & k & & . \\ & & x & l & k & & . \\ & & & l & k & & . \\ r & r & r & k & r & r \\ & & & & & x & . \\ & & & & & & x \\ & & & k & & & \\ & & & k & & & \\ & & & k & & & \end{pmatrix}, \tag{8.2}$$

where elements denoted by k and l were originally in the corresponding columns of $P\tilde{U}Q$, and those denoted by r were originally in the k-th row. The subdiagonal elements r are eliminated by

a forward sweep (Section 4.1), and the elements k below the triangle are eliminated by a single column elimination (Section 7.2).

This form of the Bartels-Golub update is conceptually the same as in LA05C (Reid, 1976, 1982) when $m = n$. By not maintaining a column list for U we lose a useful feature of Reid's implementation, wherein the row spike with elements r (see (8.2)) can often be made *shorter* by further alterations to P and Q. (Thus, Reid's forward sweep tends to add fewer nonzeros to L and U.) However, we believe that the penalty for not including this feature is usually slight (see the results of Section 9).

8.2. Adding a column. A new column c is always added to the end of A:

$$\bar{A} = (A \quad c) = L (U \quad v).$$

To perform this update, we solve $Lv = c$, insert v in U, and perform a column elimination (if $m > n$). This is a subset of the operations involved in replacing a column.

8.3. Deleting a column. Here the complementary subset is required: removal of a column of U, a cyclic permutation, and a final forward sweep. An additional task before the permutation is to reduce the column indices by one for all nonzeros to the "right" of the deleted column. This requires a scan of all rows of U.

8.4. Adding a row. This update (the simplest) can be expressed as

$$\bar{A} = \begin{pmatrix} A \\ r^T \end{pmatrix} = \begin{pmatrix} L & \\ & 1 \end{pmatrix} \begin{pmatrix} U \\ r^T \end{pmatrix}.$$

Once the given vector r is packed as a new row of U, a single forward sweep completes the task.

8.5. Replacing a row. If the i-th row is to be replaced, the new matrix may be written as $\bar{A} = A - e_i(a_i - r)^T$. The rank-one procedure of Section 8.7 is therefore used. (If the old row is not supplied, it is first recovered as $a_i = A^T e_i$; see Section 6.2.)

Row replacement could be performed more easily on a factorization of the form $NA = U$, if N were maintained explicitly as a square matrix (sparse or dense). A certain row of N could then be discarded during the update, and the old row a_i would not be needed. Difficulty arises in our case because L is held in product form.

8.6. Deleting a row. In this case, the desired modification is expressed by $\bar{A} = A - e_i a_i^T$, which effectively replaces the i-th row by zero. As above, the general rank-one procedure is used, and the old row must be supplied or computed.

Unfortunately, it is not known how to reduce the row dimension of the LU factors by 1, again because L is held in product form. As a convenience and partial solution (of nontrivial cost), we renumber the indices in L to permute the zero row to the bottom of $\bar{A}$.

8.7. Rank-one modification. For a given scalar σ and vectors v and w, this change is given by

$$\bar{A} = A + \sigma v w^T = L(U + \sigma c w^T) = L \begin{pmatrix} c & U \end{pmatrix} \begin{pmatrix} \sigma w^T \\ I \end{pmatrix}$$

where $Lc = v$. From a backward sweep we obtain the factorization $\begin{pmatrix} c & U \end{pmatrix} = \hat{L}\hat{U}$ where the first column of $\hat{U}$ is a unit vector, say βe_l, and the remainder of $\hat{U}$ is upper triangular except for row l. Thus, $\hat{U} = \begin{pmatrix} \beta e_l & \hat{U} \end{pmatrix}$ and

$$\bar{A} = L\hat{L}(\hat{U} + \beta \sigma e_l w^T) = L\hat{L}\tilde{U}$$

where $\tilde{U}$ is again upper triangular except for row l. A forward sweep eliminates that row and completes the modification.

In practice the vectors w and c are likely to be sparse, and it is worthwhile curtailing the backward sweep in the following way. Ignoring the permutations P and Q, suppose that the first nonzero of w^T is in column k and the last nonzero of c is in row l. If c_1 comprises the first k elements of c, we have

$$c = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}, \qquad U + \sigma c w^T = \left\{ U + \sigma \begin{pmatrix} c_1 \\ 0 \end{pmatrix} w^T \right\} + \sigma \begin{pmatrix} 0 \\ c_2 \end{pmatrix} w^T,$$

and the summation in braces can be performed without destroying the triangularity of U. As a result, the backward sweep need be applied only to c_2. (In fact if $k \geq l - 1$ it can be skipped altogether.) In one application, this observation roughly halved the rate of increase of nonzeros in L.

Rank-one modification to LU factors has previously been studied by Gille and Loute (1982), but their proposal does not have the stability properties of the method just described.

9. Computational results

Most of the computational results described here were derived from the set of linear programming models listed in Table 1. The first three have been used as test problems elsewhere (e.g., Reid, 1982). All are available via *Netlib* (Dongarra and Grosse, 1985).* The scaling noted for the last two problems is the default scaling performed by MINOS 5.0.

Table 1
Test problem statistics

Problem	Rows	Columns	Nonzeros	Scaled
STAIR	357	467	3857	No
SHELL	537	1775	4900	No
BP	822	1571	11127	Yes
PILOT	1460	3652	43645	Yes

The simplex method for solving such problems works with a nonsingular "basis matrix" composed of a subset of the columns. STAIR and PILOT have a staircase structure that leads to rather dense *LU* factors. SHELL is a network problem, for which it is known that all basis matrices are triangular. BP has "dual angular" *structure and requires an unusually high number* of simplex iterations, but its *LU* factors are quite sparse.

We have also experimented with the $E(n,c)$ class of matrices, as described by Østerby and Zlatev (1983). These are symmetric, positive definite matrices of order n similar to those obtained by discretizing the Laplacian operator; they have five bands of elements $(-1,-1,4,-1,-1)$ at distances $(-c,-1,0,1,c)$ from the main diagonal.

All runs were performed in double precision on an IBM 3081K (relative precision 2.2×10^{-16}). The source code was compiled with the IBM Fortran 77 compiler VS FORTRAN, using the options NOSDUMP, NOSYM and OPT(3). The linear programming runs were made using MINOS 5.0, with various routines being substituted in turn to *Factor* and *Update* the basis.

In the following sections, LU1FAC and LU2FAC denote the factorization procedures of Sections 5.1 and 5.6. LU8RPC refers to the column replacement (Bartels-Golub) procedure (Section 8.1) and LU8RPR denotes the procedure for row replacement (Section 8.5).

9.1. Factorization and solve procedures. From each linear program in Table 1, a typical square basis matrix B was selected and factorized. We then solved two linear systems $Bx = b$ and $B^T\pi = c$, as required by the simplex method. Table 2 shows the nonzero counts and the computation times (in milliseconds). The stability tolerance $\bar{\mu} = 10$ was used for each *Factor*, and the storage provided was enough to hold approximately the same number of nonzeros.

---

* For details, send electronic mail to *netlib@anl-mcs* or to *research!netlib* saying "*send index from lp/data*".

The first result for MA28 was obtained with the usual Markowitz search of both rows and columns. The second used the search strategy of examining the p shortest rows (and no columns), with $p = 10$. For LU1FAC the search of rows and columns was terminated after p consecutive ties were encountered, with $p = 10$ throughout. (Little difference was observed in a few trials with $p = 20$, 30 and 50.)

The ordering obtained by LU1FAC was used to test LU2FAC with the same matrix B. Slight differences in L and U are to be expected, since LU2FAC may perturb the row ordering for local sparsity reasons. If the stability tolerance were altered, or if the nonzeros in B were changed, a greater difference between LU1FAC and LU2FAC would be likely.

Table 2

Factor and *Solve* results for LP test problems

	STAIR		SHELL		BP		PILOT	
Problem features								
Dimension	357		537.		822		1460	
B nonzeros	3386		1490		4777		18834	
LU nonzeros								
LA05	4641		1490		6437		42598	
MA28	4644		1490		6490		43120	
MA28, $p = 10$	5041		1490		7521		53050	
LU1FAC	4652		1490		6441		48037	
LU2FAC	4648		1490		6437		48222	
Factor time								
LA05	402		39		538		15119	
MA28	284		61		586		10047	
MA28, $p = 10$	262		60		377		7247	
LU1FAC	259		101		422		7419	
LU2FAC	106		30		266		2219	
Solve times								
LA05	8	3	4	3	13	8	86	45
MA28	7	4	5	4	13	8	61	42
MA28, $p = 10$	8	5	5	4	14	10	88	51
LU1FAC	6	3	4	4	10	9	58	45
LU2FAC	7	4	4	4	11	10	71	57

The following observations are based on the results summarized in Table 2.

1. LA05 was noticeably faster than the other Markowitz routines on SHELL, because it processes triangular matrices essentially in-place (as does LU2FAC). Conversely, LA05 was significantly slower than the other *Factor* routines on the denser problems STAIR and PILOT.

2. On all problems except SHELL, the second MA28 option was faster than the first but produced rather dense *LU* factors with correspondingly higher *Solve* times.

3. LU1FAC was significantly faster than LA05 and the first MA28 option, except on problem SHELL (where repacking the columns of *L* and the rows of *U* into pivotal order probably accounts for much of the difference). On STAIR, BP and PILOT, the increases in speed of LU1FAC compared to the next best code were about 10%, 27% and 35% respectively.

4. The results for LU2FAC show the efficiency of the forward sweep procedure.

Additional experiments on the LP problems were also performed. A third MA28 option was tried on PILOT, requesting that the matrix be reduced to block-triangular form prior to *LU* factorization of each block. This led to 5% fewer nonzeros in the factors but a 5% higher *Factor* time, compared to the first MA28 option. When LU1FAC was applied to B^T from PILOT, the *LU* nonzeros were reduced to 42957 (comparable to LA05 and MA28), and the *Factor* time was reduced by about 17% to 6162. This change occurs because the pivot strategy of LU1FAC is essentially the transpose of that in LA05 and MA28. Table 2 shows that the *Solve* procedures also performed differently on B and B^T. (In linear programs, the right-hand-side vectors b are typically more dense than the vectors c.)

For the $E(n; c)$ matrices, we performed tests with $n = 800$ and $c = 4, 44, 84, 124, 164, 204$, as in Østerby and Zlatev (1983). Symmetry was preserved as expected, and the importance of curtailing the Markowitz search on matrices with regular structure (Section 5.2) was obvious. The choice of $p = 10$ as the tie limit was satisfactory on these examples also; it could evidently be "hard-wired" into the procedure for general and regular matrices alike. Elsewhere, two separate statements were executed by far the most: those locating the pivot row and pivot column in the ordered lists P and Q. Østerby and Zlatev economize in this area by updating the inverse permutations as well—a significant aid on regular matrices for a moderate increase in workspace.

Table 3 gives factorization statistics for the tie-breaking rule TB4 (see Section 5.3; note that on positive-definite matrices such as these, TB4 is equivalent to TB2). The total time required to factorize the six matrices was 4.8 seconds. Similar results were obtained for rule TB3, except that the total factorization time increased to 6.3 seconds if ties were recognized only when a smaller multiplier was found.

9.2. Update procedures. In order to test the backward sweep procedure, the simplex method was implemented by factorizing $B^T = LU$ and replacing a row of the matrix at each simplex iteration, using LU8RPR (the row replacement procedure of Section 8.5). We would not recommend this approach in practice, but our results show that row replacement can be carried out with a

Table 3

Factors of $E(800,c)$ with tie-break rule TB4, $p = 10$

c	LU nonzeros	$\max |\mu_k|$	$\min |U_{jj}|$
4	7168	.97	.08
44	20424	.49	1.5
84	15896	.44	1.8
124	12096	.45	1.9
164	10496	.45	2.1
204	8738	.38	2.3

least tolerable efficiency. In fact (much to our surprise) it proved to be more efficient than our implementation of column replacement on the simplest problem SHELL.

In simplex codes, a *Factor* is typically followed by k *Update*s, then a new *Factor* followed by k *Update*s, and so on. Tables 4 and 5 compare LA05 with LUSOL over a series of simplex iterations, with LU8RPC denoting the usual factorization and updating of B, and LU8RPR the same for B^T. Table 4 gives the average time for one factorization and k updates. The factorization frequencies were $k = 50$ for all problems except SHELL, where $k = 100$. Table 5 gives the numbers of nonzeros in the initial factorization and after i updates, where $i = 20, 30, 40$ and 50. For interest, Table 5 also shows the number of nonzeros that would be produced by the classical Product-Form (PF) update (Orchard-Hays, 1968), starting with LU factors of B^T and using them as factors of B itself.

Table 4

Average time (in seconds) for one *Factor* and k *Update*s

	STAIR ($k = 50$)	SHELL ($k = 100$)	BP ($k = 50$)	PILOT ($k = 50$)
LA05	1.91	1.42	2.45	25.1
LU8RPC	1.32	1.72	2.29	15.2
LU8RPR	2.31	1.65	2.75	20.0

As in Section 9.1, LA05 was significantly faster than the other methods on SHELL, largely because of the additional permutations in Reid's implementation of the Bartels-Golub update, which effectively maintains $L = I$ and $U = B$. The LUSOL procedures kept U as sparse as B, but during 100 iterations the updates to L increased the total nonzeros by about 50%.

Table 4 shows that LA05 and LUSOL with column replacement performed equally well on BP, but LUSOL showed a substantial advantage on STAIR and PILOT. Part of this is the result of fewer compressions of the U-file: only two or three on average between factorizations, compared

to about 20 for LA05.

 Table 5 shows that all three *LU* procedures perform more efficiently than the PF update in terms of total nonzeros. (However, the PF update has immense advantages with regard to ease of implementation.)

Table 5

Factor and *Update* nonzeros for LP test problems

		STAIR	SHELL	BP	PILOT
B_0		3540	1492	4777	18834
LA05	0	5307	1492	6437	42598
	20	6313	1492	7093	45217
	30	6648	1493	7433	46028
	40	6767	1495	7602	47460
	50	7126	1492	7713	48290
	100		1493		
LU8RPC	0	5385	1492	6441	48037
	20	6020	1835	7103	50129
	30	6429	1914	7383	50629
	40	6625	1964	7550	52063
	50	6723	2013	7652	52842
	100		2189		
LU8RPR	0	5347	1492	6514	42957
	20	9757	1590	8560	59907
	30	12472	1636	9940	68541
	40	15093	1709	11295	82693
	50	17043	1751	11610	96809
	100		1996		
PF	0	5347	1492	6514	42957
	20	12225	1866	17501	66535
	30	15662	2076	22927	78268
	40	19101	2313	28375	89869
	50	22564	2461	33328	101531
	100		3152		

10. Conclusions

We have described the salient features of a set of procedures for maintaining triangular factors of a sparse matrix, and demonstrated their practical efficiency on a representative range of problems. The Markowitz procedure LU1FAC appears to be competitive with existing codes on general problems, and to be acceptably efficient on regularly structured matrices (as measured by performance on one class of problem, $E(n,c)$). Timings have not been compared with other *Factor* routines in the latter case, and the performance of the *Update* procedures has not been studied on regular matrices.

During the last few years, LUSOL has been applied to very large matrices of the form

$$\begin{pmatrix} 0 & J \\ J^T & H \end{pmatrix}$$

arising in an early version of a sequential quadratic programming algorithm for solving optimization problems in the electrical power industry (Burchett, 1984). Consecutive column and row updates were employed to preserve symmetry. The promise of stability from bounding the multipliers throughout was consistently borne out in practice.

While column updates appear to be more efficient generally, we have shown for the first time that *LU* factors of sparse matrices can be updated with respectable efficiency following row replacement and/or *rank-one modification*. The *techniques* described here should be particularly useful for solving sequences of related linear equations.

Acknowledgements

We are grateful to Robert Burchett, Floyd Chadee, Robert Entriken, Patrick McAllister, Thomas Rutherford and John Stone for their helpful comments on various procedures from LUSOL. We also thank two referees for their valuable suggestions.

Special acknowledgement

We would like to pay tribute to Professor Gene H. Golub for his work during the last two decades on the updating of matrix factorizations. He has inspired the development of many stable numerical techniques of the kind described here.

References

Bartels, R. H. (1971). A stabilization of the simplex method, *Numerische Mathematik* 16, 414–434.

Bartels, R. H. and Golub, G. H. (1969). The simplex method of linear programming using the *LU* decomposition, *Communications of the Association for Computing Machinery* 12, 266–268.

Bartels, R. H., Stoer, J. and Zenger, Ch. (1971). A realization of the simplex method based on triangular decompositions, Contribution I/11 in *Handbook for Automatic Computation, Volume II: Linear Algebra* (J. H. Wilkinson and C. Reinsch, eds.), Springer-Verlag, Berlin.

Björck, Å. (1976). Methods for sparse least squares problems, in J. R. Bunch and D. J. Rose (eds.), *Sparse Matrix Computations*, Academic Press, London and New York, 177–199.

Burchett, R. C. (1984). Private communication, Electric Utility Systems Engineering Department, General Electric Company, Schenectady, New York.

Curtis, A. R. and Reid, J. K. (1971). The solution of large sparse unsymmetric systems of linear equations, *J. Institute of Mathematics and its Applications* 8, 344–353.

Dantzig, G. B. (1963). *Linear Programming and Extensions*, Princeton University Press, Princeton, New Jersey.

Dodson, D. S. and Lewis, J. G. (1985). Proposed sparse extensions to the basic linear algebra subprograms, *SIGNUM Newsletter* 20, 1, 22–25.

Dongarra, J. J., Du Croz, J., Hammarling, S. and Hanson, R. J. (1985). A proposal for an extended set of Fortran basic linear algebra subprograms, *SIGNUM Newsletter* 20, 1, 2–18.

Dongarra, J. J. and Grosse, E. (1985). Distribution of mathematical software via electronic mail, *SIGNUM Newsletter* 20, 3, 45–47.

Duff, I. S. (1977). MA28 – a set of Fortran subroutines for sparse unsymmetric linear equations, Report AERE R8730, Atomic Energy Research Establishment, Harwell, England.

Erisman, A. M., Grimes, R. G., Lewis, J. G. and Poole, W. G., Jr. (1985). A structurally stable modification of Hellerman-Rarick's P^4 algorithm for reordering unsymmetric sparse matrices, *SIAM J. on Numerical Analysis* 22, 369–385.

Foster, L. V. (1986). Rank and null space calculations using matrix decomposition without column interchanges, *Linear Algebra and its Applications*, to appear.

Gill, P. E., Murray, W., Saunders, M. A. and Wright, M. H. (1984). Sparse matrix methods in optimization, *SIAM J. on Scientific and Statistical Computing* 5, 562–589.

Gill, P. E., Murray, W., Saunders, M. A., Tomlin, J. A. and Wright, M. H. (1985). On projected Newton barrier methods for linear programming and an equivalence to Karmarkar's projective method, Report SOL 85-11, Department of Operations Research, Stanford University.

Gill, P. E., Murray, W., Saunders, M. A. and Wright, M. H. (1986). A note on nonlinear approaches to linear programming, Report SOL 86-7, Department of Operations Research, Stanford University.

Gille, P. and Loute, E. (1982). Updating the *LU* Gaussian decomposition for rank-one corrections; application to linear programming basis partitioning techniques, Cahier No. 8201, Séminaire de Mathématiques Appliquées aux Sciences Humaines, Facultés Universitaires Saint-Louis, Brussels, Belgium.

Hellerman, E. and Rarick, D. C. (1971). Reinversion with the preassigned pivot procedure, *Mathematical Programming* 1, 195-216.

Hellerman, E. and Rarick, D. C. (1972). The partitioned preassigned pivot procedure (P^4), in D. J. Rose and R. A. Willoughby (eds.), *Sparse Matrices and Their Applications*, Plenum, New York, 67-76.

Karmarkar, N. (1984). A new polynomial-time algorithm for linear programming, *Combinatorica* 4, 373-395.

Markowitz, H. M. (1957). The elimination form of the inverse and its applications to linear programming, *Management Science* 3, 255-269.

Murtagh, B. A. and Saunders, M. A. (1983). MINOS 5.0 user's guide, Report SOL 83-20, Department of Operations Research, Stanford University.

Orchard-Hays, W. (1968). *Advanced Linear-Programming Computing Techniques*, McGraw-Hill, New York.

Østerby, O. and Zlatev, Z. (1983). *Direct Methods for Sparse Matrices*, Lecture Notes in Computer Science 157, Springer-Verlag, Berlin.

Peters, G. and Wilkinson, J. H. (1970). The least-squares problem and pseudo-inverses, *Computer Journal* 13, 309-316.

Reid, J. K. (1976). Fortran subroutines for handling sparse linear programming bases, Report AERE R8269, Atomic Energy Research Establishment, Harwell, England.

Reid, J. K. (1982). A sparsity-exploiting variant of the Bartels-Golub decomposition for linear programming bases, *Mathematical Programming* 24, 55-69.

Saunders, M. A. (1979). Sparse least squares by conjugate gradients: a comparison of preconditioning methods, pp. 15-20 in *Computer Science and Statistics: 12th Annual Symposium on the Interface* (J. F. Gentleman, ed.), University of Waterloo, Waterloo, Ontario, Canada.

Sherman, A. H. (1978). Algorithms for sparse Gaussian elimination with partial pivoting, *ACM Transactions on Mathematical Software* 4, 330-338.

Sorensen, D. C. (1984). Analysis of pairwise pivoting in Gaussian elimination, Report MCS-TM-26, Argonne National Laboratory, Argonne, Illinois.

Todd, M. J. and Burrell, B. P. (1985). An extension of Karmarkar's algorithm for linear programming using dual variables, Report 648, School of Operations Research and Industrial Engineering, Cornell University, New York.

Wilkinson, J. H. (1965). *The Algebraic Eigenvalue Problem*, Oxford University Press, London.

Wilkinson, J. H. (1979). CS237A class notes, Computer Science Department, Stanford University.

Wolfe, P. (1962). The reduced-gradient method, unpublished manuscript, RAND Corporation.

Zlatev, Z. (1980). On some pivotal strategies in Gaussian elimination by sparse technique, *SIAM J. on Numerical Analysis* 17, 18–30.

Zlatev, Z., Wasniewski, J. and Schaumburg, K. (1981). Y12M – *Solution of Large and Sparse Systems of Linear Equations*, Lecture Notes in Computer Science 121, Springer-Verlag, Berlin.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ARo 21592.11-mA	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) MAINTAINING LU FACTORS OF A GENERAL SPARSE MATRIX		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Philip E. Gill, Walter Murray, Michael A. Saunders and Margaret H. Wright		8. CONTRACT OR GRANT NUMBER(s) N00014-85-K-0343 DAAG29-84-K-0156
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Operations Research - SOL Stanford University Stanford, CA 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR-047-064
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research - Dept. of the Navy 800 N. Quincy Street Arlington, VA 22217		12. REPORT DATE May 1986
		13. NUMBER OF PAGES 28 pp.
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office) U.S. Army Research Office P.O. Box 12211 Research Triangle Park, NC 27709		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale;
its distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Sparse matrix   rank-one modification
LU factors   Fortran software
matrix updates

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

SEE ATTACHED.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73

SOL 86-8:   MAINTAINING LU FACTORS OF A GENERAL SPARSE MATRIX, Philip E.
Gill, Walter Murray, Michael A. Saunders and Margaret H. Wright,
May 1986

We describe a set of procedures for computing and updating an LU factoriza-
tion of a sparse matrix A, where A may be square (possibly singular) or
rectangular. The procedures include a Markowitz factorization and a Bartels-
Golub update, similar to those of Reid (1976,1982). The updates provided are
addition, deletion or replacement of a row or column of A, and rank-one
modification. (Previously, column replacement has been the only update
available.)

Various design features of the implementation (LUSOL) are described, and
computational comparisons are made with the LA05 and MA28 packages of Reid
(1976) and Duff (1977).

END

D T I C

8 — 86